HUGE: An Efficient and Scalable Subgraph Enumeration System

Zhengyi Yang[1], Longbin Lai[2], Xuemin Lin[1], Kongzhang Hao[1], Wenjie Zhang[1]

[1] The University of New South Wales, [2] Alibaba Group

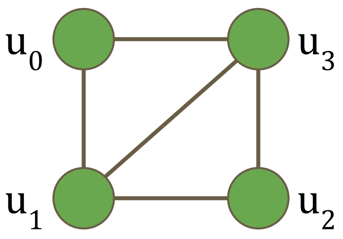
SIGMOD 2021

Outline

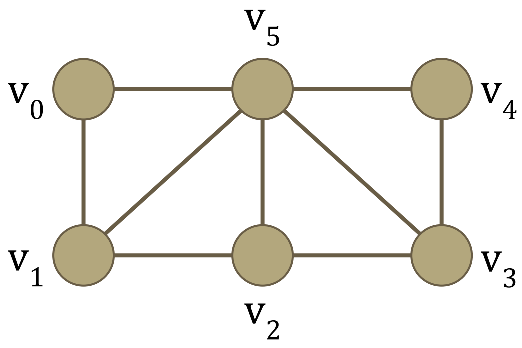
- Introduction
- The HUGE system
  - Advanced Execution Plan
  - The HUGE Compute Engine
  - DFS/BFS-adaptive Scheduler
- Experimental Evaluation
- Conclusion

Problem Definition

Subgraph Enumeration: Given a *query graph* q and a *data graph* G (both are undirected and unlabelled), the problem is to find all subgraph instances (matches) g' in G, that are isomorphic to q.



Query Graph q



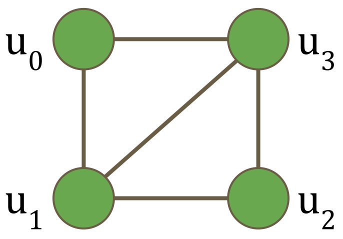
Data Graph G

Matches:

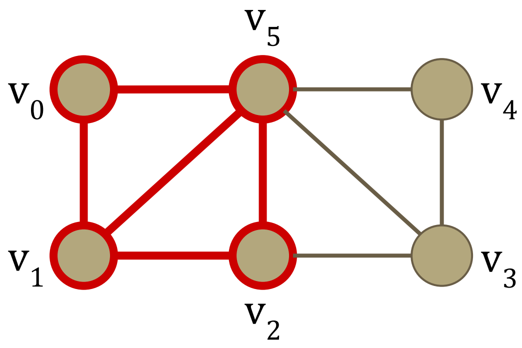
1.
2.
3.

Problem Definition

Subgraph Enumeration: Given a *query graph* q and a *data graph* G (both are undirected and unlabelled), the problem is to find <u>all</u> subgraph instances (matches) g' in G, that are <u>isomorphic</u> to q.



Query Graph q



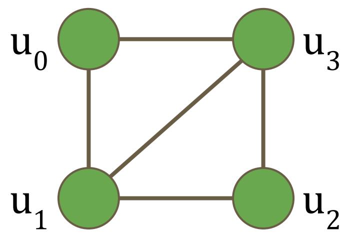
Data Graph G

Matches:

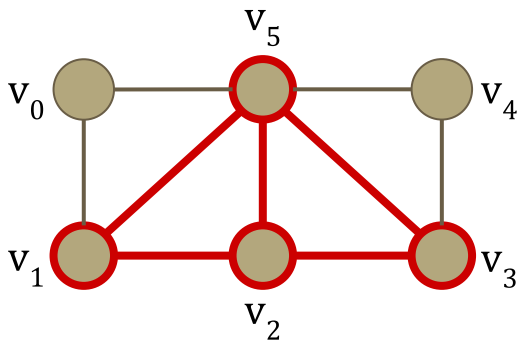
1. (u_0, u_1, u_2, u_3) -> (v_0, v_1, v_2, v_5)
2.
3.

Problem Definition

Subgraph Enumeration: Given a *query graph* q and a *data graph* G (both are undirected and unlabelled), the problem is to find all subgraph instances (matches) g' in G, that are isomorphic to q.



Query Graph q



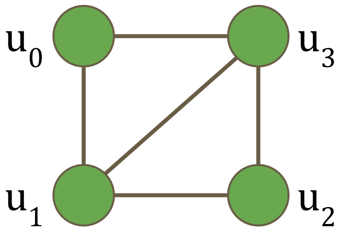
Data Graph G

Matches:

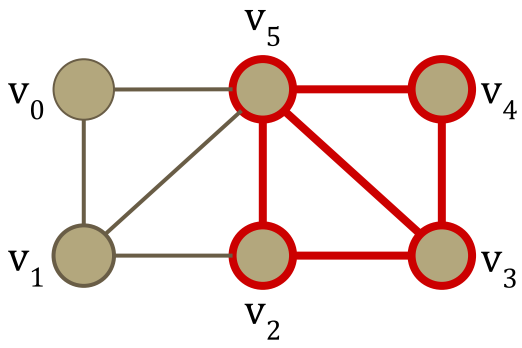
1. $(u_0, u_1, u_2, u_3) \text{ -> } (v_0, v_1, v_2, v_5)$
2. $(u_0, u_1, u_2, u_3) \text{ -> } (v_1, v_2, v_3, v_5)$
3.

Problem Definition

Subgraph Enumeration: Given a *query graph* q and a *data graph* G (both are undirected and unlabelled), the problem is to find all subgraph instances (matches) g' in G, that are isomorphic to q.



Query Graph q



Data Graph G

Matches:

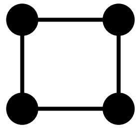
1. $(u_0, u_1, u_2, u_3) \text{ -> } (v_0, v_1, v_2, v_5)$
2. $(u_0, u_1, u_2, u_3) \text{ -> } (v_1, v_2, v_3, v_5)$
3. $(u_0, u_1, u_2, u_3) \text{ -> } (v_2, v_3, v_4, v_5)$

Existing Works

- **Join-based Algorithms**
  - Use distributed joins to compute matches (with different join algorithms and join orders)
  - *Push* data (intermediate results) from the host to remote machines
  - **High tension on both communication and memory usage**
- **Pull-based Algorithms**
  - *Pull* (and *cache*) the data graph instead to reduce communication volume and memory consumption
  - **May not reduce computation and communication time**

Initial Experiment - Setup

We conduct an initial experiment of representative existing works.

- Dataset:
  - Query Graph: `Square` 
  - Data Graph: `LiveJournal` (4.8 million vertices, 43.4 million edges)
- Algorithms:
  - Join-based
    - SEED: Binary join algorithm with optimal bushy plan
    - BiGJoin: Worst-case optimal join algorithm
  - Pull-based
    - BENU: Store the data graph in external distributed key-value database and run backtracking (DFS) as in a single machine
    - RADS: Expand-star*-and-verify in a pulling manner

*Star: a tree of depth 1

Initial Experiment - Results

Comm. Mode	Work	Total Time (s)	Comp. Time (s)	Comm. Time (s)	Comm. Volume (GB)	Peak Mem (GB)
Pushing	SEED	1536.6	343.2	1193.4	537.2	42.3
	BiGJoin	195.9	122.1	73.8	534.5	14.3
Pulling	BENU	4091.7	3763.2	328.5	25.3	1.3
	RADS	2643.8	2478.7	165.1	452.7	19.2
Hybrid	**HUGE**	**52.3**	**51.5**	**0.8**	**4.6**	**2.2**

High communication volume and memory consumption

High external overhead and low utilisation

Sub-optimal plans

- Efficiency and scalability are jointly determined by:
  - **Computation**, **Communication** and **Memory management**
- **None of the works achieves satisfactory performance for all three perspectives**

Challenges

- **Execution Plan**
  - Previous works achieve “optimality” in a *specific* context
  - None can guarantee the best performance by all means
- **Communication Mode**
  - Non-trivial to make pull-based communication efficient
  - An efficient plan may require *both* pushing and pulling
- **Scheduling Strategy**
  - DFS strategy can lead to low hardware utilisation while BFS strategy has high memory demands
  - *Static heuristics* all lack in a tight bound and can sometimes perform poorly in practice

Contributions

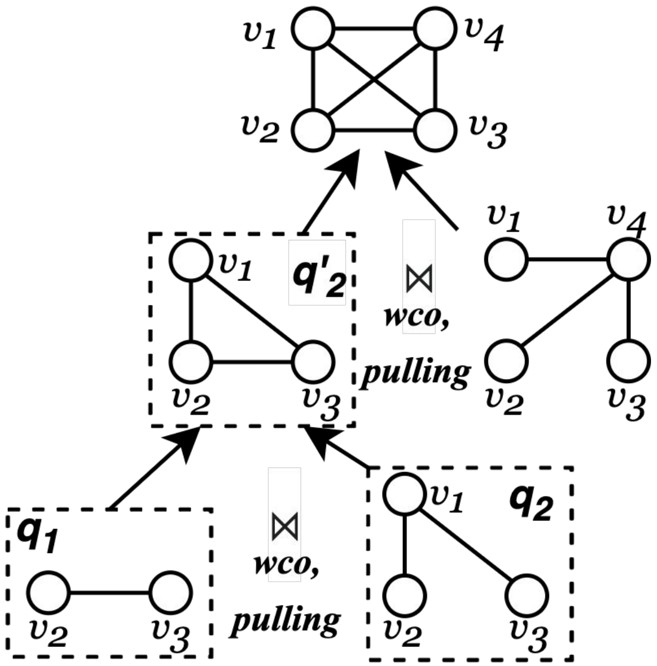
HUGE is a pushing/pulling-**H**ybrid s**U**b**G**raph **E**numeration system that features:

- **Advanced execution plan**
  - **Optimal** execution plan in a more **generic** context
- **Pushing/pulling-hybrid compute engine**
  - Efficiently support **both** push-based and pull-based communication
- **DFS/BFS-adaptive scheduler**
  - **Bounded-memory** execution without sacrificing computing efficiency

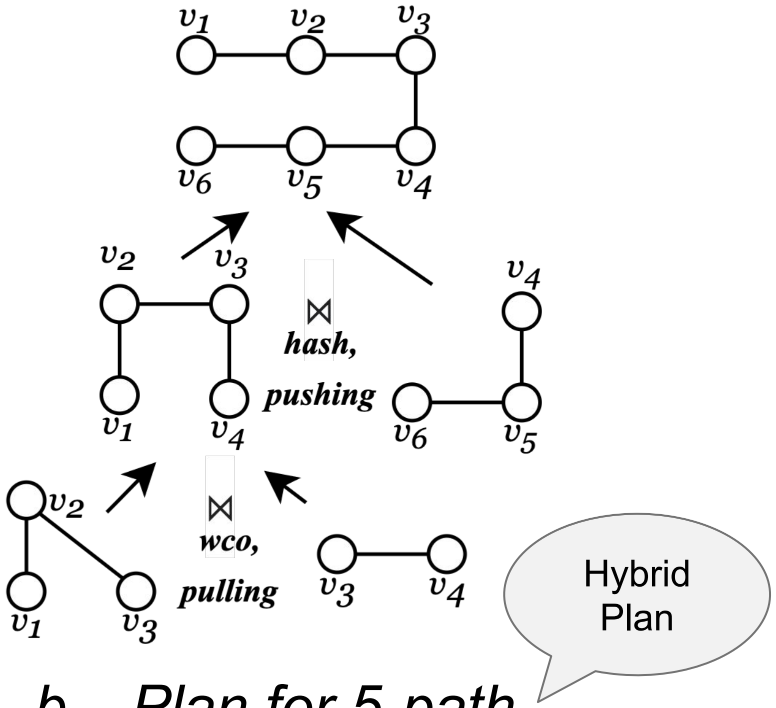
Advanced Execution Plan

- Break down an execution plan into **logical** and **physical** aspects
  - A unified logical join-based framework: $R(q) = R(q_1) \bowtie R(q_2) \bowtie \cdots \bowtie R(q_k)$
    - Join Unit: edges, stars, ~~cliques~~
    - Join Order: left-deep, bushy
  - Physical join processing:
    - Join Algorithm: hash join, worst-case optimal (wco) join
    - Communication Mode: pushing, pulling
- Dynamic programming based optimiser to minimise both *communication* and *computation* in **generic** context

Example HUGE Plans



a. Plan for 4-clique



b. Plan for 5-path

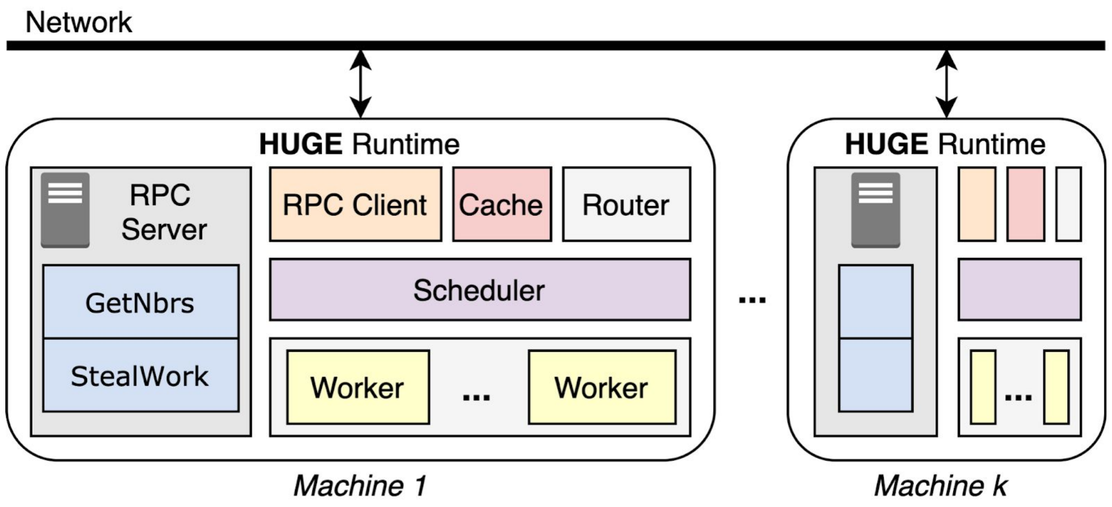
All existing works can be readily plugged in to enjoy automatic performance improvement

HUGE Compute Engine

- Adopt the popular **dataflow model** for distributed execution
  - Execution plans are translated into dataflow graphs using different HUGE operators
- Pushing/pulling-hybrid **dual communication** mode
  - A new cache policy with two-stage execution strategy
- **Dynamic work stealing** for better load balancing
  - Two-layer intra- and inter- machine mechanism

System Architecture

- **RPC Server/Client:** Serve pulling requests
- **Router:** Pushes data to other machine
- **Worker:** Run de-facto computation
- **Cache:** HUGE's LRBU cache
- **Scheduler:** HUGE's DFS/BFS adaptive scheduler



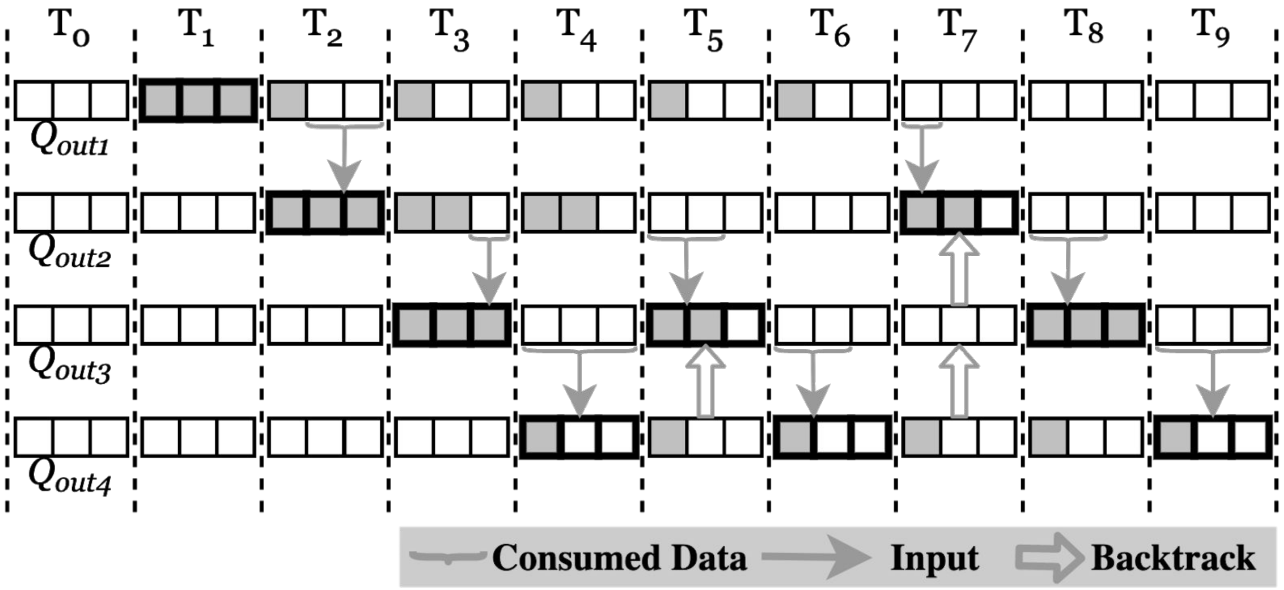
UNSW SYDNEY

LRBU Cache

- Two vital issues of traditional cache (e.g. LRU or LFU)
  - Memory copies
  - Locks
- Least recent-batch used (LRBU) cache
  - Target at a **zero-copy** and **lock-free** cache access
  - **Two-stage execution strategy**
    - Fetch stage: aggregate remote vertices, send async pull requests in bulk, and write remote vertices to the cache => Write-only (using single writer)
    - Intersect stage: read cache and compute intersections => Read-only
  - Synchronisation cost <7.5% with performance improvement 4.4x on average comparing with concurrent LRU

DFS/BFS-adaptive Scheduler

- Each dataflow operator is equipped with a **fixed-size output queue**
- Adopts **BFS-style** scheduling whenever possible to fully leverage parallelism
- Adapts dynamically to **DFS-style** scheduling if the output queue is full



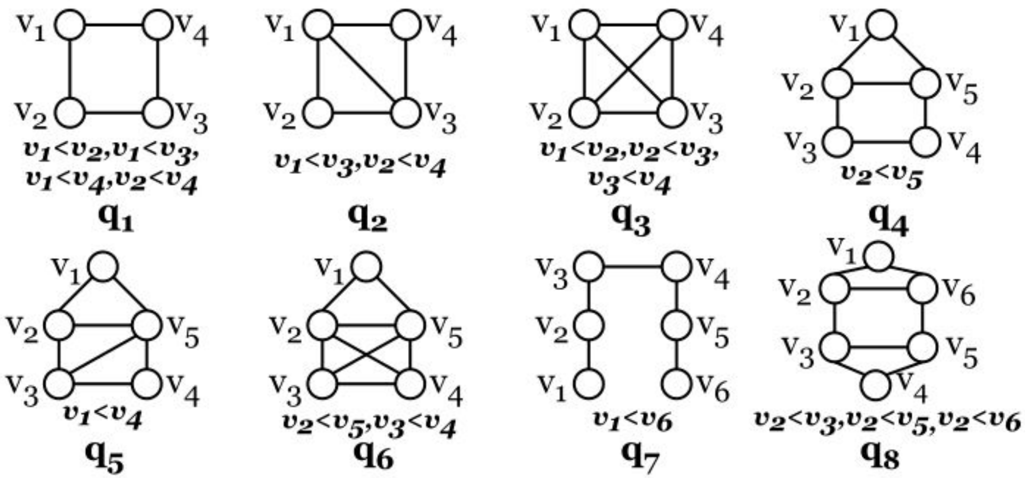
Experimental Evaluation

- **Hardware:**
  - Local cluster: 10 machines with 4-core Intel Xeon E3-1220, 64G memory, 1TB Disk, connected on a 10Gps network
  - AWS cluster: 16 AWS “r5.8xlarge instances” with 32 vCPUs, 256G memory, 1TB EBS storage, 10Gps network (for the web-scale experiments only)
- **Datasets :**
  - 7 real-world data graphs, 8 queries selected from prior works
- Others:
  - Cache size: 30% of the data graph
  - Allow 3 hour maximum running time for each query

Datasets

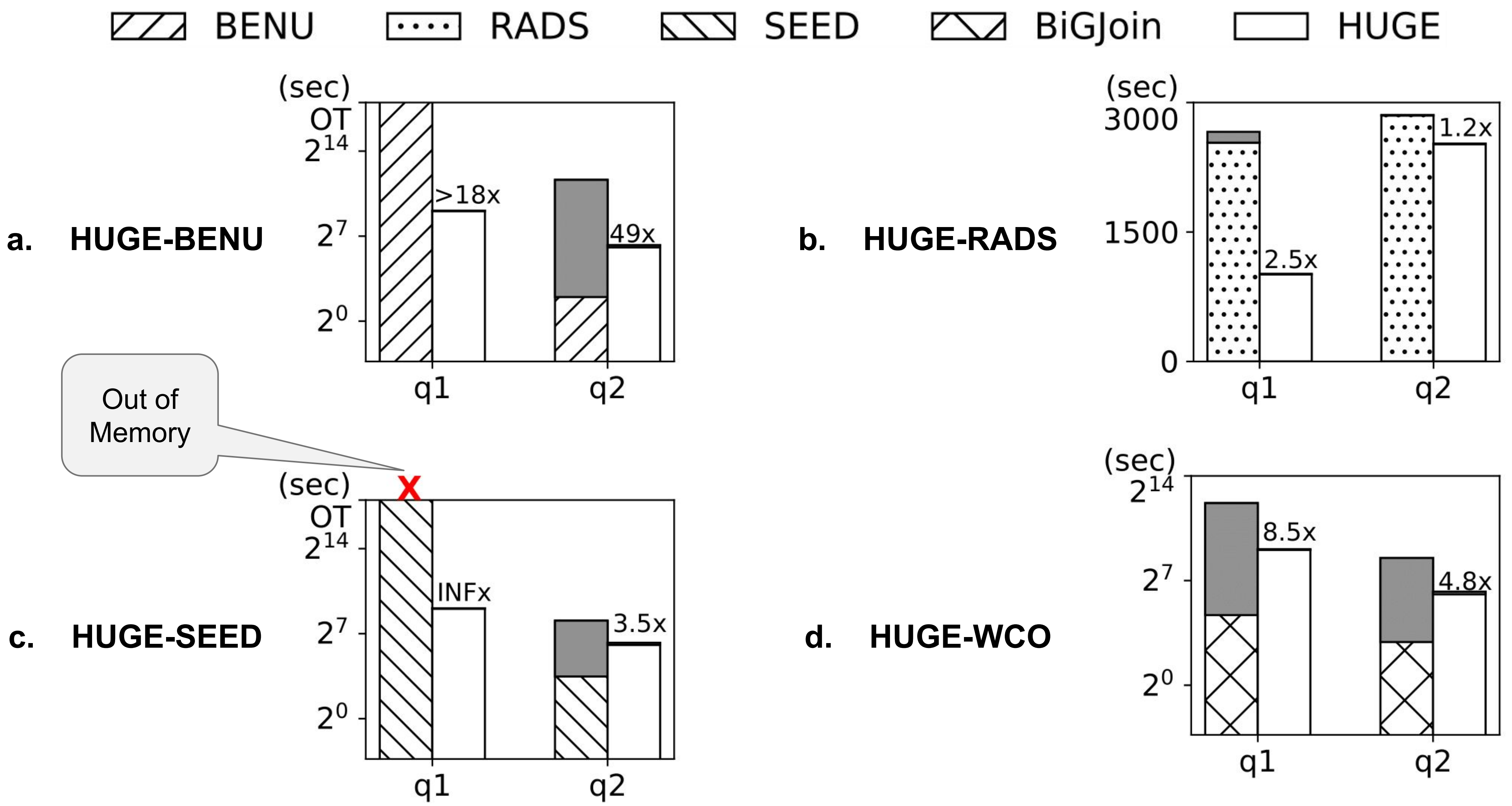
	|V|	|E|
Google (GO)	875,713	4,322,051
LiveJounal (LJ)	4,847,571	43,369,619
Orkut (OR)	3,072,441	117,185,083
UK02 (UK)	18,520,486	298,113,762
EU-road (EU)	173,789,185	347,997,111
Friendstar (FS)	65,608,366	1,806,067,135
ClueWeb12 (CW)	978,409,098	42,574,107,469

a. Table of Data Graphs

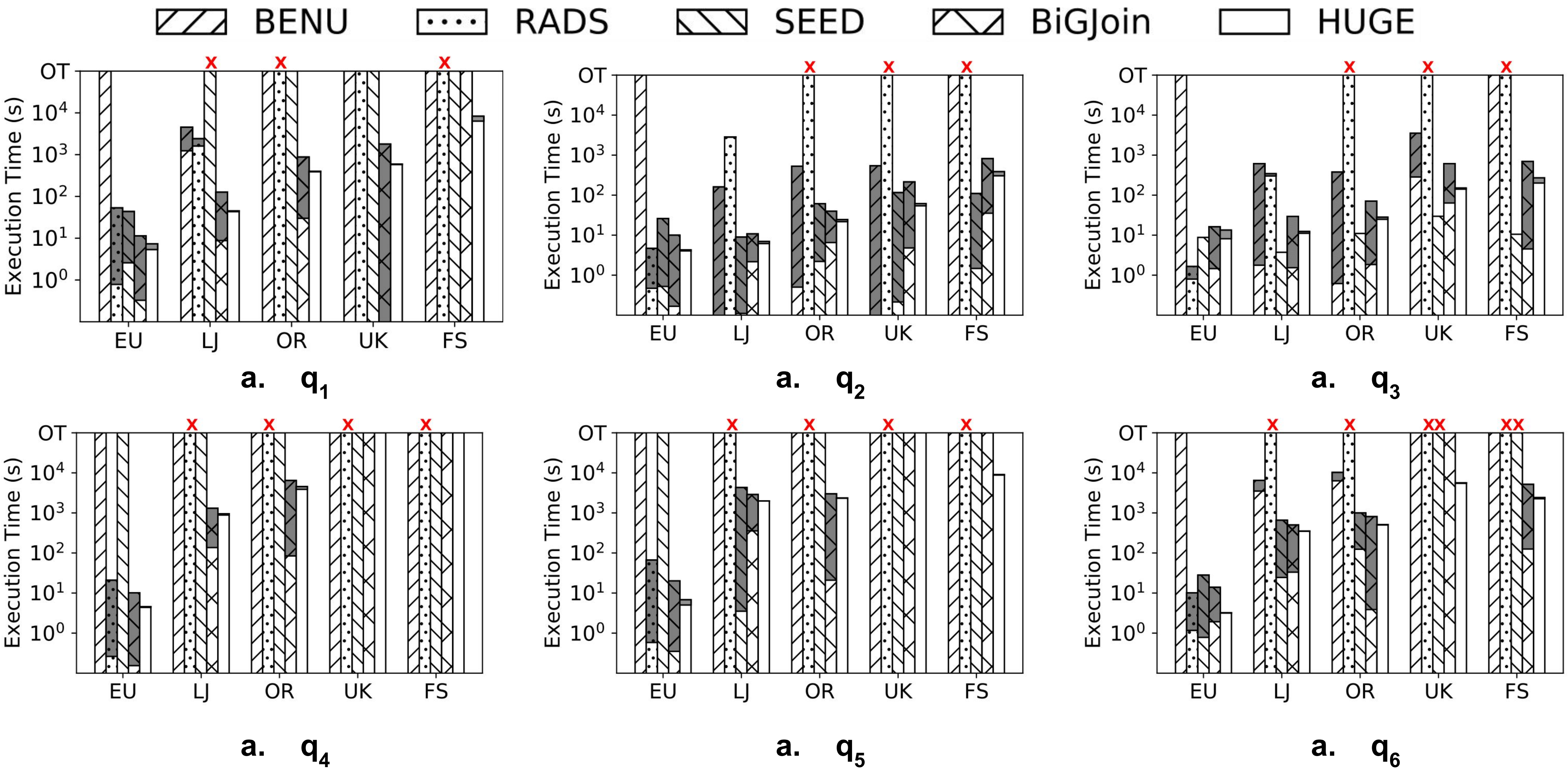


b. Query Graphs

Speed Up Existing Algorithms (on LJ)



All-Round Comparisons



a. q_1

a. q_2

a. q_3

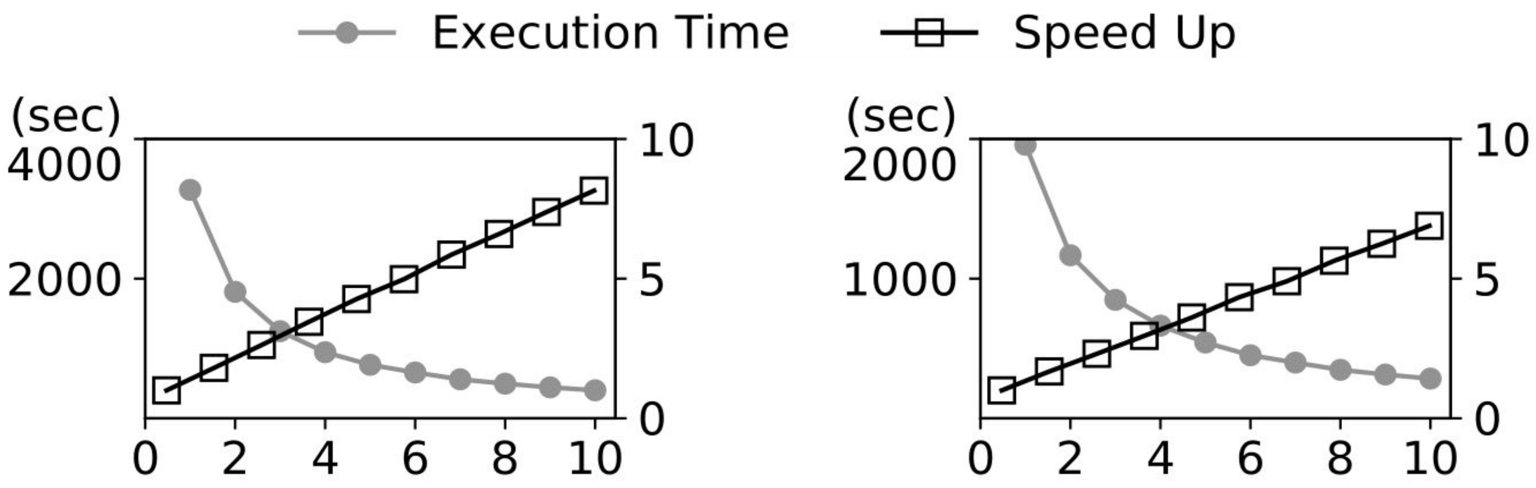
a. q_4

a. q_5

a. q_6

Scalability

- Vary Number of Machines (on FS)



a. $\mathbf{q_2}$ **b.** $\mathbf{q_3}$

- Web-scale Graph (on CW)

	q_1	q_2	q_3
Throughput	2,895,179,286/s	354,507,087,789/s	206,696,071/s

Conclusion

- HUGE is an efficient and scalable subgraph enumeration system in the distributed context
- HUGE is designed to be flexible for extending more functionalities such as:
  - Cypher-based Distributed Graph Databases
  - Graph Pattern Mining (GPM) Systems
  - Shortest Path & Hop-constrained Path

Thanks!